LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-681427

libPowerMon: A Low-overhead Profiling Framework for Correlating Program Context with System-Level Metrics

A. P. Marathe, J. Yeom, H. Gahvari, A. Bhatele

January 29, 2016

High Performance Power Aware Computing 2016 Workshop at International Parallel and Distributed Processing Symposium 2016
Chicago, IL, United States
May 23, 2016 through May 27, 2016

Disclaimer

libPowerMon: A Lightweight Profiling Framework to Profile Program Context and System-level Metrics

Aniruddha Marathe, Hormozd Gahvari, Jae-Seung Yeom, Abhinav Bhatele

Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, California 94551 USA
Email: {marathe1, gahvari1, yeom2, bhatele}@llnl.gov

***Abstract*—As power becomes one of the most important resources to provision while building modern HPC systems and applications, it becomes crucial to obtain deeper insights into applications' power and thermal characteristics. There exists a need to correlate application context with processor-level and system-level power and thermal measurements. Existing profiling tools to monitor power and thermal measurements either operate at a granularity that is not fine enough to correlate with application-level events that describe application context or are not equipped to sample application-level events. In this work, we introduce *libPowerMon*, a lightweight user-level profiling framework to simultaneously sample user-specified application events and system-level metrics at up to 1 kHz sampling intervals. At the application level, *libPowerMon* provides a source-level phase markup interface to capture application context. It records MPI and OpenMP events, and samples processor state at a finer temporal granularity. At the system level, *libPowerMon* samples power and thermal characteristics and provides an interface to set processor and DRAM power. We present three case studies that demonstrate the benefits of *libPowerMon* in saving cluster-level power and improving application performance within a system-enforced power limit.**

***Keywords*-profiling; program context; power; thermal measurements; performance**

I. Introduction

As high performance computing (HPC) architectures approach exascale, power is becoming a critical operating constraint rather than an optimization goal. Power bounds imposed by the limitations of existing infrastructure as well as the stakeholders drive the need to develop new techniques to extract maximum performance at limited power. Achieving optimal performance on power-constrained resources requires a fine-grained understanding of power usage and performance profile of the application. Without source-level modifications, existing profiling methods may be extended to provide *aggregated* rather than fine-grained node-level power and performance characteristics of the application. Detailed profiling of application-level power and performance characteristics requires potentially expensive and tedious source-level modifications. Given the limitations of existing profiling tools to capture unique power and performance profiles of individual phases of an application, it becomes difficult to allocate power to critical parts of the application to improve performance.

To improve upon these limitations, we present *libPowerMon*, a phase-level power and performance sampling framework. *libPowerMon* samples, at a configurable sampling interval, the following state of the application: current timestamp, MPI calls, OpenMP events and call site, hardware performance counters, and processor and DRAM power use. *libPowerMon* also provides additional capabilities to identify logical phases demarcated through simple phase markup routines for post-processing. *libPowerMon* also consists of independent software components to capture node-level sensor data to primarily provide a deeper understanding of the phase-level power-usage characteristics of HPC applications at scale. With the help of this tool, we have been able to shorten the gap between node-level power draw and processor and DRAM power usage.

In order to demonstrate the benefits of correlating application context with system-level metrics, enabled by *libPowerMon*, we present three case studies. First, *libPowerMon* enabled us to observe the temporally non-deterministic nature of certain computation phases of the ParaDiS code [1]. Correlating power measurements with program context guided us in re-defining computation phases around power usage signatures instead of logical function boundaries. Such findings about the nature of computation phases of production applications such as ParaDiS are crucial towards improving the effectiveness of a power-optimizing run-time system. Second, analysis using *libPowerMon* showed that fans on our cluster nodes ran at full speed settings regardless of the processor temperature at various power limits. By adjusting the inefficient fan speed settings on each node, we saved over 15 kW of power at the cluster level.

Finally, we used *libPowerMon* to measure the impact of application configuration options on phase-level and node-level characteristics of linear solver applications. We found that a trade-off exists for different linear solver options in terms of power usage and performance subject to a global power constraint for two problems: *27-point 3D Laplacian* and *Convection-diffusion*. Aggregate power measurements of important computation phases showed that for a system-enforced power limit, selecting a superior algorithm and configuration based on empirical data results in up to 15% improvement in application performance.

II. Motivation

Several application profiling tools exist that help application developers and users record and characterize application behavior by profiling application-level events. These profiling

tools can transparently collect application phases marked by the user and can typically record aggregate events associated with these phases. For example, it is possible to record attributes such as aggregate power usage or cumulative performance counters between successive application-level events such as MPI calls. Due to the arbitrary nature of such events captured by profiling tools, the gaps between successive data points tend to be arbitrary, which makes it difficult to extract instantaneous values of the metrics being captured. Moreover, such profiling tools suffer from high run-time overhead as the recording or in-situ processing of events potentially occurs on the critical path of the application.

On the other end of the spectrum are system-level measurements tools, typically operated as independent hardware modules (on-board sensors, third party power and temperature meters, etc.). These profiling tools perform sampling-based profiling of system-level metrics at a high sampling rate. However, these tools suffer from lack of program context in order to draw meaningful observations from the collected data. Also, such modules are either extremely difficult or impossible to install at scale due to high aggregate costs or the sensitive nature of codes running on clusters at national laboratories such as LLNL. Therefore, in order to correlate application context with system-level power and relevant metrics, it becomes necessary to develop infrastructure that can sample and record at these levels. *libPowerMon* presented in this work enables profiling of this kind for the first time on our clusters.

III. Application- and System-level Profiling Infrastructure

This section describes our design approach and implementation details of the profiling framework components. First, we provide an overview of the different processor- and node-level profiling tools that we employ in *libPowerMon* to characterize phase-level power and performance characteristics of the application. Second, we discuss our node-level sampling component. Finally, we describe the design and implementation of our sampling-based phase-level profiling library to capture user-annotated phases, MPI calls, OpenMP events, user-defined MSR counters and power usage information.

A. Monitoring Tools and Interfaces

To monitor application phases and system-level metrics, we employed following existing monitoring interfaces and tools.

IPMI interface: The Intelligent Platform Management Interface (IPMI) specification defines a set of interfaces for platform management that include sensor monitoring, system event monitoring, power control, and serial-over-LAN (SOL) [2]. We used *ipmi-sensors* tool to record current readings of sensors available on the node hardware.

LibMSR: *libMSR* is a user-level interface to several model-specific registers in Intel processors [3]. We used *libMSR* to record hardware performance counters, effective frequency, temperature, and processor and DRAM power draw on each compute node.

TABLE I
IPMI data collected by *libPowerMon*.

Entity	IPMI field	Description
Node power	PS1 Input Power	Power supply 1 input power
Node current	PS1 Curr Out	Power Supply 1 Max. Current Output
Node voltage	BB [12.0V|5.0V|3.3V]	Baseboard +12V|+5V|+3.3V
	BB 1.5 P[1-2]MEM	Baseboard processor memory voltage
	BB 1.05Vccp P[1-2]	Baseboard processor voltage
Node thermal	BB P[1-2] VR Temp	Processor voltage regulator temperature
	Front Panel Temp	Front panel temperature
	SSB Temp	Server South Bridge temp.
	Exit Air Temp	Exit air temperature
	PS1 Temperature	Power supply 1 temperature
Processor thermal	P[1-2] Therm Margin	Processor thermal margin
	P[1-2] DTS Therm Mgn	Processor DTS thermal margin
	DIMM Thrm Mrgn [1-4]	DIMM Thermal Margin
Node air flow	System Airflow	Volumetric airflow in CFM
	System Fan [1-5]	Fan speeds in RPM

PMPI profiling layer: We used the PMPI profiling layer to initiate and terminate our sampling framework and to capture MPI event entry and exit at run-time. With the PMPI profiling layer the sampling library we can intercept MPI calls through static or dynamic linking with the application without introducing direct source-level changes.

OpenMP tools: We used OpenMP tools interface to record entry into and exit from OpenMP parallel regions [4]. We introduced OpenMP callbacks to log meta data associated with each OpenMP region invocation such as OpenMP region ID, call site and stack back-trace.

Using the monitoring interfaces, we developed a two-level sampling framework to sample system-level metrics and application-level context. First, we developed a node-level component to record system-level sensor data available on the compute nodes on the clusters at Lawrence Livermore National Laboratory (LLNL). Second, we developed a sampling library that is invoked at run-time during application initialization to sample application context and processor-level hardware performance counters. This is the first time such two-level sampling framework has been deployed on LLNL clusters.

B. Node-level Component: IPMI Recording Module

On LLNL clusters, reading IPMI sensor data requires root access which severely limits access to node-level sensors. We developed software components to enable IPMI profiling for regular users on these clusters. The software components include a job scheduler plug-in that is invoked after the compute resources have been allocated but before the job has been started. A sampling script then samples IPMI data through *freeIPMI* [2] interface in the background. The sampled data on all compute nodes along with UNIX timestamp is funneled into one sampling log that is prefixed with the job ID and compute node ID for convenient post-processing. Table I

enumerates an interesting subset of sensor data that we capture on the compute nodes on LLNL clusters.

C. Application-level Component: Sampling Library

Figure 1 shows the system model with various application-level, processor-level and node-level components of *libPowerMon*. *libPowerMon* links with the application transparently through the PMPI profiling layer. After MPI_Init(), *libPowerMon* initializes the sampling environment based on the user-specified configuration defined through the environment variables. *libPowerMon* initializes the headers in the main trace file and an optional per-process file to report instances of single or nested application phases between successive samples. The format of the trace file is described in Table II. Due to the overhead associated with on-line processing of the data to be sampled, *libPowerMon* records some of the data online and stores the rest of the data in the memory for post-processing offline.

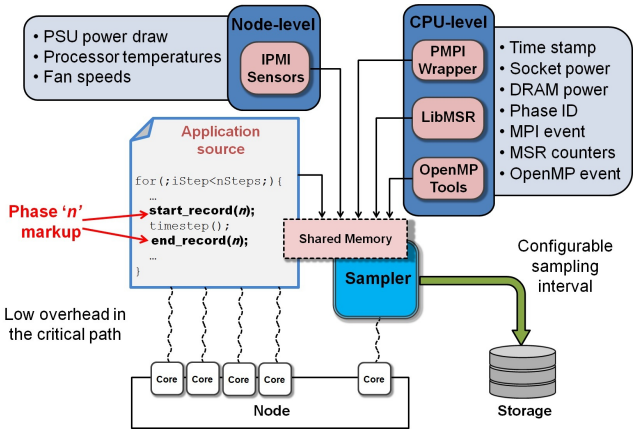


Figure 1. System model describing node-level, processor-level and source-level components of *libPowerMon*.

Phase markup interface: *libPowerMon* provides a minimal, low-overhead interface to the user for source-level phase markup annotations. Through the interface, each interesting application phase can be assigned an ID, and the start and end of the phase can be specified. The phase markup functions log entry or exit of a phase along with a timestamp. The sampling library post-processes the log to derive phase-stack information and appends it to the trace.

Sampling instantaneous application state: The primary profiling component of *libPowerMon* is a dedicated thread to sample application performance metrics. The sampling thread is spawned at the end of MPI_Init() and it is pinned to the largest core ID to minimize its interference with the application. The number of MPI processes assigned to one sampling thread can be configured at initialization. The sampling logic uses UNIX shared memory interface to read the sampled data recorded by each MPI process after MPI_Init(). The sampling logic uses LibMSR to record user-specified MSRs, APERF/MPERF, Time Stamp Counter (TSC), derived processor temperature, derived processor power usage and DRAM power usage. The sampling logic also records the UNIX timestamp in seconds (to allow merging of the sampled data with the IPMI data at post-processing) and a per-process timestamp in milliseconds relative to MPI_Init() initialization time.

TABLE II
APPLICATION-LEVEL AND SYSTEM-LEVEL DATA SAMPLED BY *libPowerMon*.

Field	Description
Timestamp.g	UNIX timestamp of a sample (seconds)
Timestamp.l	Relative timestamp of the sample since MPI_Init() (milliseconds)
Node ID	Node ID of MPI process
Job ID	Job ID of MPI process
Phase ID	A list of phases (as demarcated in the application source) that appeared in a sampling interval
MPI_start, MPI_end	MPI event log including entry and exit timestamp, calling phase ID and MPI-specific information
Hardware counters	User-specified hardware performance counters
Temperature	Processor temperature data
APERF, MPERF	Hardware performance counters to calculate effective processor frequency
Power usage	Processor and DRAM power draw (watts)
Power limits	User-defined processor and DRAM power limits (watts)

Issues in data collection: The logic for on-line phase-stack sampling and processing along with MPI event profiling introduced unwanted overheads in the execution of the sampling thread. At one-millisecond granularity, phase-stack sampling and MPI event logging produced a large trace data especially for applications with high phase counts and MPI operations. This stalled the sampling thread at arbitrary intervals and introduced non-uniformity in the sampling interval. Our investigation revealed that the stalls also happened due to write buffer flushes by the operating system at arbitrary intervals. To resolve this issue, we enabled partial buffering of trace data for minimizing the size of in-memory trace as well as the size of write buffer. We introduced post-processing logic to process phase stack and MPI event profile in the MPI_Finalize PMPI handler. These significantly reduced the on-line overhead in the sampling thread and resolved the issue of non-uniformity in sampling.

Overheads: We measured the overhead of running MPI applications with our sampling framework with two different settings: 1) no MPI process was bound and 2) an MPI process was bound to the sampling thread core. We measured the overhead for an application with over 50 nested phases and generated over a 100 MPI events every few seconds. We set sampling frequencies between 1 Hz and 1 kHz. When no MPI process bound to the sampling thread core, *libPowerMon* introduced less than 1% overhead in execution time even at 1 kHz sampling frequency. When an MPI process was bound to the sampling thread core, *libPowerMon* introduced between 1% to 5% overhead in execution time.

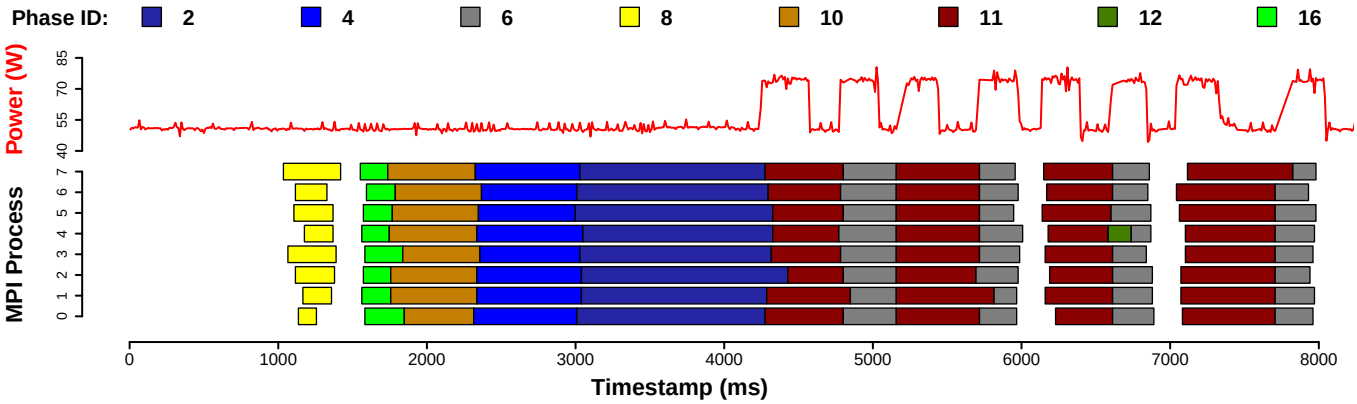


Figure 2. Progress and power usage of several phases of the ParaDiS application running with 8 MPI processes on a processor. The overlap of power usage over phase boundary of phase 11 shows the granularity at which the phase boundaries must be revised.

IV. Experimental Setup and Measurements

We deployed our sampling framework on two LLNL clusters: Catalyst and Cab. Catalyst is a 324-node Intel Xeon E5-2695 Ivy Bridge cluster with an InfiniBand QDR interconnect. Each Catalyst node is composed of two 12-core processors and 128 GB of DRAM. Cab is a 1296-node Intel Xeon E5-2670 cluster with an InfiniBand QDR interconnect. Each Cab node is composed of two 8-core processors and 32 GB of DRAM. We have evaluated the sampling library component on both clusters, but the installation of our IPMI recording component has been confined to Catalyst due to lack of root-level permissions on Cab. Therefore, the case studies presented in the rest of the paper are based on our experiments solely on the Catalyst cluster.

For benchmarking, we used FT and EP applications from the NAS Parallel Benchmark Suite [5], ParaDiS [1] and CoMD applications [6]. NAS EP is a primarily computation-bound application ideal for testing power characteristics of a platform. NAS FT and CoMD applications have varying degrees of compute, memory and communication boundedness. ParaDiS [1] is a production dislocation dynamics simulations application that operates on unbalanced, dynamically changing data set sizes across MPI processes. The random nature of data set sizes results in non-determinism and varying computational load across MPI processes.

For our first case study with ParaDiS, we used a modified version of the "Copper" input set provided with ParaDiS with 100 timesteps. We ran ParaDiS at 16 MPI processes with 8 MPI processes on each processor of the Catalyst node. We fixed the processor power limit to 80 watts. For our second case study, we configured NAS EP and FT with class C input size at 16 MPI processes. We chose class C input size to ensure long enough execution times for our tests. We also chose CoMD as our third application for which we chose input problem size of 50x50x50 with 100 timesteps. We ran all applications on a single node with 8 MPI processes on each processor. We varied the processor power limit from 30 watts to 90 watts in steps of 5 watts. For our third use case, we used the *new_ij* application provided with the *HYPRE* library. We ran *new_ij* with *27-point Laplacian* and *Convection-diffusion* problem configurations at 8 MPI processes with 2 MPI processes on each node (1 process on each processor). We varied the number of OpenMP threads from 1 to 12. Section VII describes the details of *new_ij* application provided with the *HYPRE* library and relevant configuration options.

V. Case Study I: Characterizing Phases of Applications with Non-Determinism: ParaDiS

This case study presents of our initial findings with ParaDiS application using *libPowerMon*. First, we demonstrate that using *libPowerMon* we were able to visually correlate, for the first time, processor-level power metrics with application phases. Second, we describe how the phase-level characteristics of ParaDiS changed our (rather simple) assumptions about how typical HPC applications behave.

A. Correlating Processor-level Power with Application Phases

Figure 2 shows a partial snapshot of execution of ParaDiS application covering 8 MPI processes on a processor. The processor-level power limit was set to 80 watts. The processor-level power usage was sampled at 100 Hz frequency. The figure shows distinct phase boundaries manually marked in ParaDiS source code and the node-level power usage of individual phases.

We make several observations from the figure. First, while some phases operate near the processor-level power limit, a major portion of the execution was spent at a low power draw near 51 watts. This information is visually useful in understanding the distribution of fine-grained power usage under a processor power limit. Second, successive invocation of some phases shows different execution times. For example, Figure 2 shows that phases 6 and 11 are invoked repeatedly, but they perform differently across invocations. Third, the processor-level power usage signature of phase 6 is different across invocations, suggesting different computation characteristics across those invocations. Finally, processor power usage within a phase shows significant variation (e.g., phase

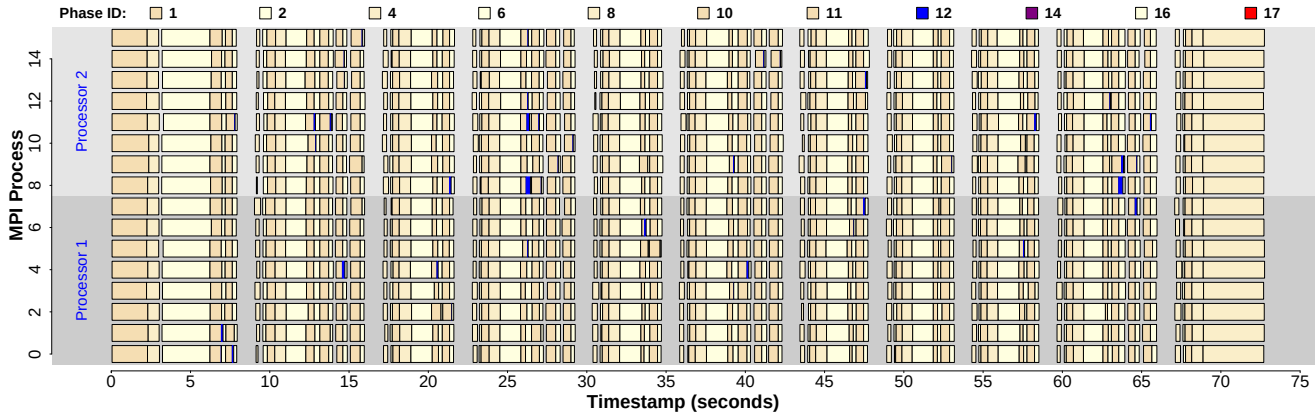


Figure 3. Progress of various phases of ParaDiS running with 16 MPI processes on one node (8 MPI processes per processor). Phases with non-deterministic occurrence are shown in darker shades.

11) which suggests that phases must be redefined beyond semantic boundaries based on power-usage characteristics. Such demarcation of phases is potentially beneficial to power-constrained run-time systems that depend on the user to outline phase boundaries for their power allocation strategies [7].

B. Identifying Phase-level Non-determinism

Figure 3 shows a full-scale run of ParaDiS at 16 MPI tasks. Repeating phases are shown in shades of light colors whereas, arbitrarily occurring phases are shown in darker colors. An example of an arbitrarily occurring phase is phase 12 (shown in blue) which appears arbitrarily in the execution path of most MPI processes. We found that the amount of time spent in phase 12 and its occurrences throughout the execution of the application are unpredictable. The presence of such arbitrarily occurring phases in the execution path shows that it is much more challenging to apply to ParaDiS the optimization techniques that rely on the repetitive nature applications. Our observations provide a compelling reason to look beyond deterministic, load-balanced proxy applications commonly used to evaluate power and performance optimization approaches.

VI. Case Study II: System-wide Power Savings through Insights into Fan Settings

The purpose of this case study is to understand, for the first time, the node-level power draw for different applications. We demonstrate that the combination of IPMI profiling (total node power) and RAPL profiling (processor and DRAM power) allowed us, for the first time, to measure static power in userspace and correlate those measurements to application execution. We used *libPowerMon* to record various metrics specified in Table II for three standard HPC benchmarks with varying processor and memory-boundedness, and a computation-bound application configured to run the processors as fast as possible. We configured our experiments at processor power limits from low to high on nodes with processors known to have low variation.

A. Initial Observations and Configuration Recommendations

Figure 4 shows our node-level and processor-level power measurements along with aggregate fan speeds for the three applications. Node power was consistently 120 watts greater than the sum of processor and DRAM power, and fan speeds remained near the maximum RPM (revolutions per minute) regardless of the amount of power being used by the application. In this case, static power was approximately 100 watts regardless of what the processor was doing. This observation was true even at lowest power limits which ran processors much cooler. The thermal headroom available to processors was between 70 °C to 50 °C for minimum and maximum power limits even for compute-bound applications. The obvious diagnosis turned out to be correct: the BIOS fan speed setting was effectively set to *performance* mode at over 10,000 RPM. Further investigation into the default BIOS fan settings on the compute nodes confirmed our diagnosis about a high fan power draw (each node houses five fans). We requested the fan setting to be altered to *auto* which, according to the server board specification controls fan speed based on instantaneous processor temperature.

Based on our recommendations, the Catalyst cluster was rebooted using a new BIOS setting. Static power dropped by at least 50 watts per node with the new fan speeds in the range of 4500-4600 RPM (>50% decrease in RPM compared to maximum). Given the 300+ compute nodes, and given that these five 20W fans per node were running at full speeds whether the node was idle or busy, we are now saving on the order of 15kW of power on this cluster alone.

Based on this data, we observed a 4 °C increase in node temperature (maximum increase of 9 °C) and a 1 °C increase in intake air temperature. Figure 5 shows the comparison between various node-level and processor-level measurements for the three applications. Out of the three applications, FT showed less than 10% performance degradation at the lowest power bounds but we have not yet been able to verify if this is within normal variation. For all three applications, the thermal headroom decreased by as much as 20 °C (60 °C to

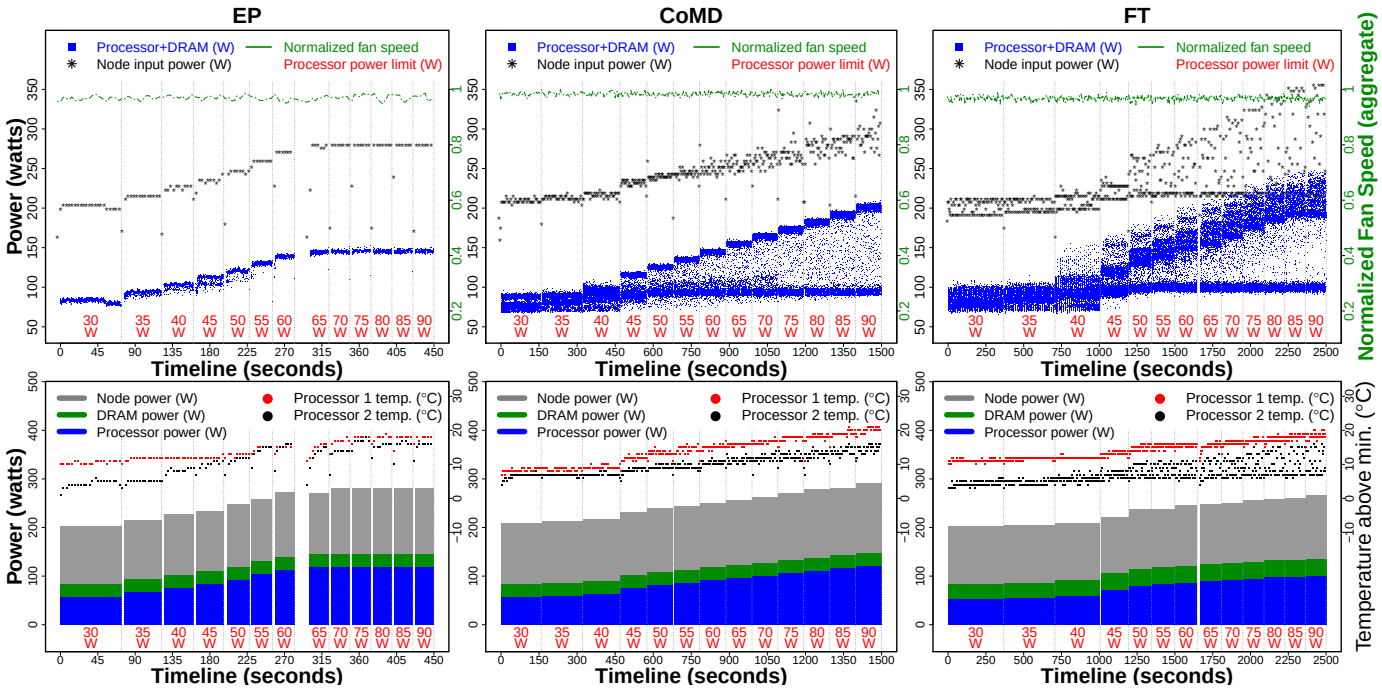


Figure 4. Observed node-level and processor-level metrics of power usage, fan speed and processor temperature at different power bounds for three applications: EP, CoMD and FT.

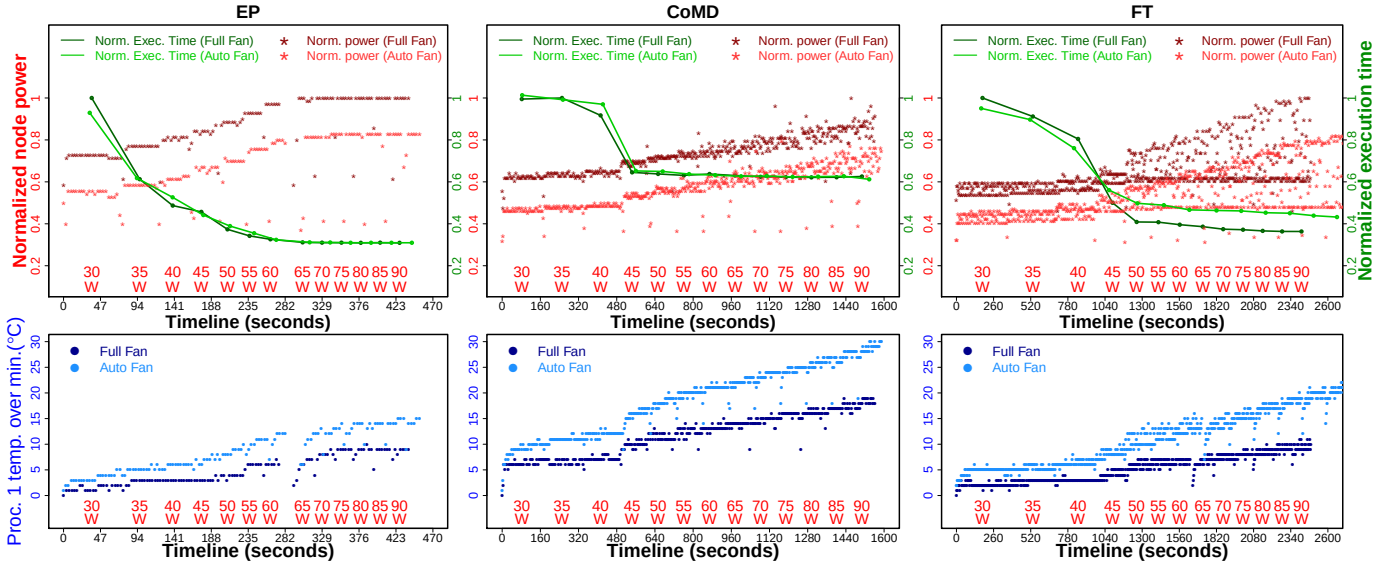


Figure 5. Comparison of change in node-level and processor-level measurements between full versus automatic fan speed settings.

40 °C). Most interesting, perhaps, is that there is still only a weak correlation between total node power and fan speeds. We suspect that the fans are still running too fast much of the time given that a significant thermal headroom still exists (the extra RPMs do not contribute any additional performance) and perhaps not running fast enough at high loads (thus reducing the effectiveness of the CPU turbo mode due to reduced thermal headroom on the processors). A strong statistical correlation between input power and processor temperatures at different power limits with automatic fan setting makes our suspicion stronger.

B. Ramifications of the New Fan Settings

Due to the cluster being a shared resource we do not have nearly as much before-and-after data as we would like. We are conscious of the fact that effects are both per-application and per-node, so we are limited to a handful of results where the same application was run on the same nodes both before and after the change in fan settings. Therefore, with the new fan settings, we now need to answer the following questions. How much of that savings is being spent with increased cooling

costs? How much has application performance been affected? Do we have the optimal fan setting for these nodes? Our ongoing work is focused on answering these questions.

VII. Case Study III: Impact of Configuration Options on Power-constrained Performance

This case study demonstrates the benefit of using *libPowerMon* to understand the power usage vs. performance characteristics of application phases for different algorithmic and data representation choices of an application. A typical scientific application (e.g. a class of linear solvers) provides several configuration options to select the algorithm, intermediate data models and the level of accuracy of the output for a given input problem configuration. Traditionally, such algorithms have been designed considering full computation power of the processor at maximum operating frequency and in terms of number of available cores. Therefore, any slowdown in the operating frequency of the processor when hardware-level power constraints are applied by the run-time system can affect the performance and consumed power of the algorithms. Given that HPC clusters are being overprovisioned for compute resources with power as the limited factor [7], it has become necessary to quantify the impact of hardware-enforced processor power limits on these algorithms that are not necessarily optimized for arbitrary processor power allocation.

Existing tools to measure aggregate power across several runs of such configurations are limited in capturing critical application context necessary to understand power and performance characteristics of these algorithms. We use *libPowerMon* with such class of applications to record program context and power usage metrics of important phases of the chosen algorithm with the selected configuration options. Using application context information, execution time and system-level power metrics, we correlate, for the first time, the effect of changing the computation algorithm and its configuration options on the power and performance characteristics of important phases in the application. We consider two candidate problems for this case study: *Convection-diffusion* and *27-point Laplacian* solved by the *new_ij* application provided with the *HYPRE* library [8]. We describe in detail our study of impact of selecting different configurations involving *HYPRE* solver options, level of concurrency and processor-level power limits on phase-level power and performance trade-offs.

A. Description of Sample Problems

new_ij is a test program distributed with *HYPRE* that allows for the evaluation of different Algebraic MultiGrid (AMG) solver parameters, such as solver type, smoother type, coarsening strategy, and interpolation scheme on a number of different test problems. In our work, we varied the solver options summarized in Table III using two different test problems:

27pt: a 3D Laplace problem discretized using a 27-point finite difference stencil on a cube.

Convection-diffusion: the steady-state convection-diffusion problem

$$-c_x u_{xx} - c_y u_{yy} - c_z u_{zz} + a_x u_x + a_y u_y + a_z u_z = 1$$

discretized using a 7-point stencil on a cube, with all c_i and a_i set to 1. Second-order centered differences are used for the second derivatives, and first-order forward differences are used for the first derivatives.

TABLE III
HYPRE solver configuration options for *new_ij*

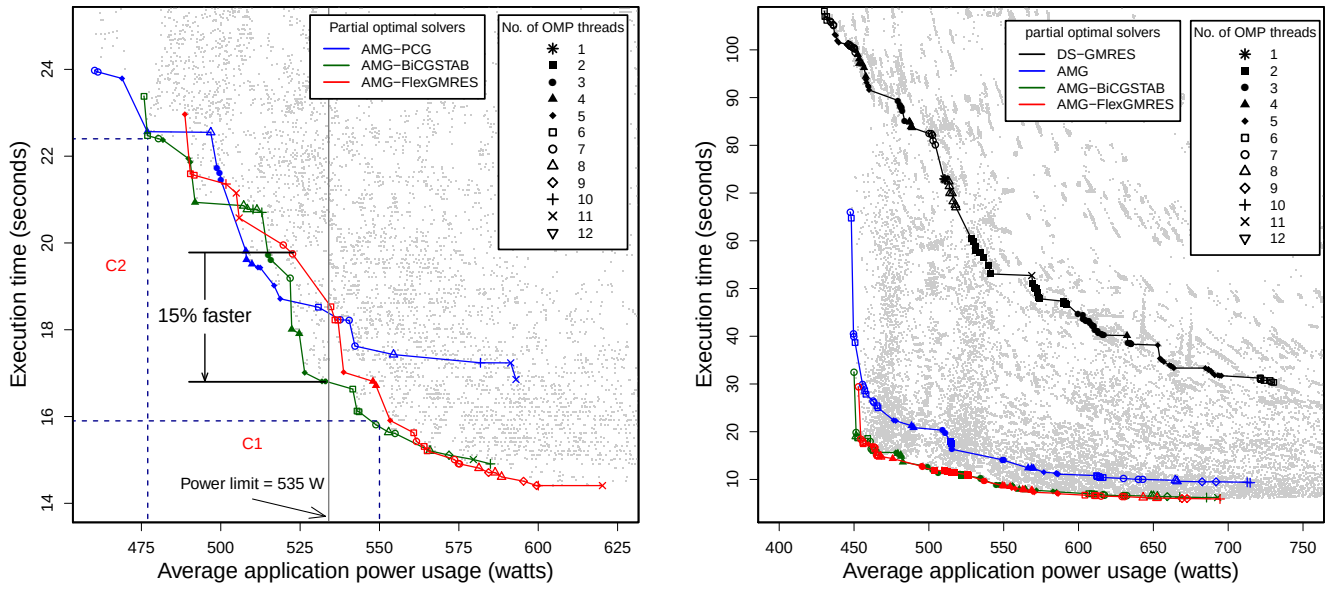
Solver	**Smoother**
AMG	Hybrid Gauss-Seidel
AMG-PCG	Hybrid backward Gauss-Seidel
DS-PCG	Forward L1-Gauss-Seidel
AMG-GMRES	Chebyshev
DS-GMRES	
AMG-CGNR	**Coarsening options**
DS-CGNR	hmis
PILUT-GMRES	pmis
ParaSails-PCG	
AMG-BiCGSTAB	**Pmx**
DS-BiCGSTAB	2
GSMG	4
GSMG-PCG	6
GSMG-GMRES	
ParaSails-GMRES	**Fixed options**
DS-LGMRES	-intertype 6
AMG-LGMRES	-tol 1e-8
DS-FlexGMRES	-agg_nl 1
AMG-FlexGMRES	-CF 0

The solver options in Table III that are allowed to vary over four different areas: solver, smoother, coarsening scheme, and interpolation operator. Additionally, there are four options that are kept fixed. Due to space limitations, we only discuss the above four options important in this case study.

The solvers considered were standalone algebraic multigrid (AMG), along with a number of different preconditioned Krylov subspace methods. In the case of AMG or solvers preconditioned with it (AMG-PCG, AMG-GMRES, AMG-CGNR, AMG-BiCGSTAB, AMG-LGMRES, and AMG-FlexGMRES), the implementation used was Hypre's own BoomerAMG solver [9]. The different Krylov solvers were preconditioned conjugate gradient (PCG), GMRES, CGNR, BiCGSTAB, LGMRES (the accelerated GMRES method of Baker, et al. [10]), and FlexGMRES (the inner-outer preconditioned GMRES method of Saad [11]). Other preconditioners used besides AMG were diagonal scaling (DS), PILUT [8], GSMG [12], and ParaSails [13].

The smoothers used are all described in [14]. Hybrid Gauss-Seidel and Hybrid backward Gauss-Seidel perform forward or backward Gauss-Seidel smoothing on-process, and Jacobi smoothing off-process. The Forward L1-Gauss-Seidel and Chebyshev smoothers are more advanced, non-hybrid smoothers, designed for large problems on machines with multicore nodes.

The coarsening options are one of two independent-set based coarsening algorithms, HMIS and PMIS, that are described in [15]. They were designed with low-complexity in mind, to enable good performance on large problems on massively parallel machines. Most modern classical AMG methods use one of these two coarsening schemes.



(a) 27-point Laplacian

(b) Convection-diffusion

Figure 6. Pareto-efficiency curve for 27-point Laplacian and Convection-diffusion problems at different average power usages.

The `-Pmx` option controls the interpolation operator, bounding the number of entries per row at the given number (2, 4, and 6 in our experiments). This is done to further reduce operator complexity and improve parallel performance.

B. Evaluation

We profiled *new_ij* on Catalyst cluster at LLNL using *libPowerMon* for *27-point Laplacian* and *Convection-diffusion* problems at eight MPI processes on four nodes (two MPI processes on a dual-processor node, one on each processor). For both problems, we exhaustively ran each combination of configuration options listed previously. For each such combination, we varied two run-time options: 1) number of OpenMP threads per processor from 1 to 12 (maximum), and 2) processor power limit from 50 watts to 100 watts through RAPL in steps of 10 watts (that is, global power limits from 400 watts to 800 watts in steps of 80 watts keeping DRAM power uncapped). For each problem, this resulted in over 62K unique combinations of configuration and run-time options.

In each run, *new_ij* executed two phases in sequence: *setup* followed by *solve*. Using phase-level application context recorded by *libPowerMon*, we extracted execution time and average power for the *solve* phase, since typical large scale runs of real applications represented by *new_ij* spend majority of computation time in the *solve* phase. Figure 6 shows aggregate power usage for different runs of *new_ij solve* phase for *27-point Laplacian* and *Convection-diffusion* problems in figures (a) and (b) respectively. Each grey data point corresponds to one combination of configuration and run-time options. We focus on a subset of solvers that were power-efficient for the system-enforced job-level power limit applied in steps. Each colored curve joins all runs of a solver that are *Pareto-efficient* in terms of average power usage and execution time. That is, all data points that belong to a solver on the other side its colored curve away from the origin are less efficient in terms of power and execution time (i.e., result in higher power usage and longer execution times). Note that, the colored configurations represent *best-case* and in practice even for each solver, it is difficult to select a combination of solver configuration and run-time options that is Pareto-efficient under a fixed processor power limit.

We make several observations regarding the power and performance characteristics of *solve* phases of *27-point Laplacian* and *Convection-diffusion* problems from the plots.

First, even in the absence of a global power limit, selecting the optimal solver configuration and run-time options is non-trivial. For example, *AMG-FlexGMRES* is the optimal solver for *27-point Laplacian* and *Convection-diffusion* problems at effectively no power limit (towards right hand bottom of the plots). However, the optimal smoother and other subset of configuration options differ for the two problems—*Chebyshev* smoother option results in better performance for *27-point Laplacian*, whereas, *Hybrid Gauss-Seidel* smoother option shows superior performance for *Convection-diffusion*. Also, the optimal number of OpenMP threads for *27-point Laplacian* and *Convection-diffusion* is 11 and 10, respectively (due to space limitations, we did not highlight smoother and coarsening configuration options in the plots).

Second, selecting the optimal solver configuration options subject to a global power limit is challenging. For example,

the optimal solver for *27-point Laplacian* and *Convection-diffusion* is *AMG-FlexGMRES* at high power limits. However, subject to lower power limits, the difference in execution times between best-case *AMG-FlexGMRES* and the absolute best-case solver configuration under that power limit can be significant.

Third, for a given solver configuration, picking the optimal number of OpenMP threads is a non-trivial problem due to non-linearity between number of OpenMP threads and global power usage. For example, in case of *27-point Laplacian*, power usage increases between 475 watts and 550 watts with a decrease in OpenMP thread count for *AMG-FlexGMRES* and *AMG-BiCGSTAB* solvers (the behavior is more chaotic for *AMG-BiCGSTAB*). We observe similar behavior with *Convection-diffusion* for *DS-GMRES* solver and specifically for low power usage configurations (between 400 watts to 475 watts) for all four solvers. Further analysis of hardware performance counters reveals that such behavior is due to degree of memory- vs. compute-boundedness of individual configuration and run-time options.

These observations show that choosing the optimal configuration subject to a user-defined or system-enforced constraint is a non-trivial problem. For a certain system-enforced global power limit, selecting a configuration known to be optimal under normal conditions (i.e., without any power constraint) may be sub-optimal. For example, consider *27-point Laplacian* with a 535 watts global power limit as indicated by the vertical grey line. The optimal solver *AMG-FlexGMRES* is 15.1% slower than *AMG-BiCGSTAB* under the global power limit. For a certain user-defined energy budget, several solver configurations exist with power vs. execution time trade-off. For example in *27-point Laplacian*, there exist several configurations under user-defined energy budget. For instance, for an energy budget of 11 kJ with *27-point Laplacian* there exist two candidate configurations C1 and C2 with energy requirements within the defined budget. Then, the optimal configuration depends on user's preference for optimizing execution time or power usage under the energy budget.

VIII. Related Work

A large body of work exists on the measurement of power at the system level, since power consumption has become a crucial metric. These approaches can be categorized as physical vs. model-based measurements with certain accuracy vs. coverage/overhead trade-offs. Physical measurements are typically performed by measuring the electrical power usage at the power supply unit using third-party power meter boards [16]–[18]. Although such interfaces can accurately track power consumption at the system-level, their cost and space overheads are prohibitive at scale and therefore such interfaces cannot be scaled beyond a few nodes. IPMI is a message-based interface designed to monitor platform status (power, temperature, voltage, fan speed, etc.) at the hardware level [19]. Using software interfaces to IPMI such as freeIPMI, OpenIPMI and IPMItool, the user can read the system-level metrics of interest. Although IPMI is typically supported on all modern cluster server boards, it must be executed asynchronously, since it operates out-of-band and typically requires root privileges which makes it difficult for regular users to use such interfaces on shared clusters. Finally, model-based power measurement functionality as part of the processor firmware provides a light-weight interface for processor power measurement e.g., Running Average Power Limit (RAPL). Model-based power measurements suffer from two important drawbacks: they are localized to the processor and they may not be accurate [3], [20].

Extensive research work has been presented on profiling application-level context and events [21]–[24] A majority of this work suffers from a localized view of the system in terms of correlating power consumption with program context. Also, most of this work is trace driven where as *libPowerMon* is sampling-based which provides fine-grained, rather than aggregate power profiles. *libPowerMon* also provides a collection of scripts to visualize these two data sets together. It may be possible to extend our work to write plug-ins for visualization tools such as Vampir and Scalasca to use them for visualization.

The work that is most related to ours is Caliper framework [25]. Caliper was designed for extensibility and provides interfaces for source-level phase markup. An important difference between *libPowerMon* and Caliper is that *libPowerMon* records sampling-based profile of application context as well as important processor-level metrics such as power, effective frequency, temperature and MSRs. Additionally, *libPowerMon* samples system-level metrics using IPMI interfaces which Caliper does not natively capture.

A large piece of literature exists that describes correlation of application context with system-level metrics combining one or more of the profiling techniques listed above. For example, Georgiou et al. present a modified SLURM installation with modules for power measurement through IPMI and RAPL [26]. Their work shows a trade-off in accuracy and sampling overhead between IPMI and RAPL-based power measurements and validate their measurements with electric power meters. Similar line of work studies the feasibility and effectiveness of several profiling techniques on different localized settings [27]–[32]. It may be argued that existing hardware and software profiling techniques can be deployed to gather the data that *libPowerMon* collects, however, setting up such an environment is non-trivial given the temporal and asynchronous nature of data collection requirements. An important issue with the profiling tools mentioned in this section is that they either fail to provide interfaces to combine application context with system-level metrics, or are capable of providing aggregate metrics rather than fine-grained measurements enabled by our sampling-based approach. For example, power profiling interfaces capture very limited or no application context. On the other hand, application profiling tools fail to capture power, thermal and other system-level metrics at a granularity that enables characterization at a given program context. Existing sampling-based monitoring tools suffer from significant profiling overhead due to frequent

interrupt processing and data collection that is always executed synchronously on the critical path of the application instead of asynchronously. To the best of our knowledge, this is the first time we have combined system-level metrics and program context at this level of granularity at low overhead. Our case studies show that combining the two levels of profiling enables us to get insights that were not possible with previously presented profiling tools.

IX. Summary and Future Work

This work demonstrated the importance of combining application context and system-level measurements to gain deeper insights into phase-level interactions between the application and the system. We plan to use these insights to enhance our understanding of how allocated resources are consumed by the application and the system. Based on phase-level performance and power characteristics, a performance-optimizing run-time system can make informed decisions about allocating limited system resources to extract more performance out of a given application. In turn, these improvements can guide the necessary hardware and software enhancements for efficiently utilizing available resources in a resource-constrained environment.

Acknowledgment

This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344 (LLNL-CONF-681427). We thank Reza Zemani and Barry Rountree for their invaluable suggestions.

References

[1] V. Bulatov, W. Cai, J. Fier, M. Hiratani, G. Hommes, T. Pierce, M. Tang, M. Rhee, K. Yates, and T. Arsenlis, "Scalable line dynamics in ParaDiS," in *Supercomputing*, 2004.

[2] freeIPMI. http://www.gnu.org/software/freeipmi/index.html. FreeIPMI Core Team.

[3] L. L. N. S. LLC. libMSR. https://github.com/scalability-llnl/libmsr.

[4] A. Eichenberger, J. Mellor-Crummey, M. Schulz, N. Copty, J. DelSignore, R. Dietrich, X. Liu, E. Loh, and D. Lorenz, "OMPT and OMPD: OpenMP Tools Application Programming Interfaces for performance analysis and debugging," Tech. Rep., 2013.

[5] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, P. Frederickson, T. Lasinski, R. Schreiber *et al.*, "The NAS parallel benchmarks summary and preliminary results," in *Supercomputing*, 1991, pp. 158–165.

[6] (2013) CoMD. https://github.com/exmatex/CoMD.

[7] A. Marathe, P. E. Bailey, D. K. Lowenthal, B. Rountree, M. Schulz, and B. R. de Supinski, "A run-time system for power-constrained HPC applications," in *International Supercomputing Conference*, 2015.

[8] R. D. Falgout and U. M. Yang, "HYPRE: A Library of High Performance Preconditioners," in *Computational Science–ICCS 2002*. Springer, April 2002, pp. 632–641.

[9] V. E. Henson and U. M. Yang, "BoomerAMG: A parallel algebraic multigrid solver and preconditioner," *Applied Numerical Mathematics*, vol. 41, pp. 155–177, 2002.

[10] A. H. Baker, E. R. Jessup, and T. Manteuffel, "A Technique for Accelerating the Convergence of Restarted GMRES," *SIAM Journal on Matrix Analysis and Applications*, vol. 26, pp. 962–984, 2006.

[11] Y. Saad, "A Flexible Inner-Outer Preconditioned GMRES Algorithm," *SIAM Journal on Scientific Computing*, vol. 14, pp. 461–469, 1993.

[12] E. Chow, "An unstructured multigrid method based on geometric smoothness," *Numerical Linear Algebra With Applications*, vol. 10, pp. 401–421, 2003.

[13] ——, "Parallel implementation and practical use of sparse approximate inverse preconditioners with a priori sparsity patterns," *International Journal of High Performance Computing Applications*, vol. 15, pp. 56–74, 2001.

[14] A. H. Baker, R. D. Falgout, T. V. Kolev, and U. M. Yang, "Multigrid Smoothers for Ultraparallel Computing," *SIAM Journal on Scientific Computing*, vol. 33, pp. 2864–2887, 2011.

[15] H. De Sterck, U. M. Yang, and J. J. Heys, "Reducing Complexity in Parallel Algebraic Multigrid Preconditioners," *SIAM Journal on Matrix Analysis and Applications*, vol. 27, pp. 1019–1039, 2006.

[16] WattsUp? Meter. https://www.wattsupmeters.com/secure/index.php.

[17] J. H. Laros, P. Pokorny, and D. DeBonis, "Powerinsight-a commodity power measurement capability," in *Green Computing Conference (IGCC), 2013 International*. IEEE, 2013, pp. 1–6.

[18] R. Ge, X. Feng, S. Song, H.-C. Chang, D. Li, and K. Cameron, "Powerpack: Energy profiling and analysis of high-performance systems and applications," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 21, no. 5, pp. 658–671, May 2010.

[19] T. Schwenkler, "Intelligent platform management interface," *Sicheres Netzwerkmanagement: Konzepte, Protokolle, Tools*, pp. 169–207, 2006.

[20] H. David, E. Gorbatov, U. Hanebutte, R. Khanna, and C. Le, "RAPL: Memory power estimation and capping," in *ACM/IEEE International Symposium on Low Power Electronics and Design*. ACM, 2010, pp. 189–194.

[21] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey, and N. R. Tallent, "HPCToolkit: Tools for performance analysis of optimized parallel programs," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 6, pp. 685–701, 2010.

[22] S. Shende and A. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 287–311, 2006.

[23] A. Knüpfer, C. Rössel, D. an Mey, S. Biersdorff, K. Diethelm, D. Eschweiler, M. Geimer, M. Gerndt, D. Lorenz, A. Malony *et al.*, "Score-p: A joint performance measurement run-time infrastructure for Periscope, Scalasca, TAU, and Vampir," in *Tools for High Performance Computing 2011*. Springer, 2012, pp. 79–91.

[24] J. Vetter and C. Chambreau, "mpiP: Lightweight, scalable MPI profiling," 2005.

[25] D. Böhme, T. Gamblin, P.-T. Bremer, O. Pearce, and M. Schulz, "Caliper: Composite performance data collection in HPC codes."

[26] Y. Georgiou, T. Cadeau, D. Glesser, D. Auble, M. Jette, and M. Hautreux, "Energy accounting and control with SLURM resource and job management system," in *Distributed Computing and Networking*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, vol. 8314, pp. 96–118.

[27] F. Moghaddam, T. Geenen, P. Lago, and P. Grosso, "A user perspective on energy profiling tools in large scale computing environments," in *Sustainable Internet and ICT for Sustainability, 2015*, 2015, pp. 1–5.

[28] H. Sharifi, O. Aaziz, and J. Cook, "Monitoring HPC applications in the production environment," in *Proceedings of the 2Nd Workshop on Parallel Programming for Analytics Applications*, ser. PPAA 2015. New York, NY, USA: ACM, 2015, pp. 39–47.

[29] D. hackenberg, T. Ilsche, R. Schone, D. Molka, M. Schmidt, and W. Nagel, "Power measurement techniques on standard compute nodes: A quantitative comparison," in *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, April 2013, pp. 194–204.

[30] M. E. M. Diouri, M. F. Dolz, O. Glück, L. Lefèvre, P. Alonso, S. Catalán, R. Mayo, and E. S. Quintana-Ortí, "Assessing power monitoring approaches for energy and power analysis of computers," *Sustainable Computing: Informatics and Systems*, vol. 4, no. 2, pp. 68–82, 2014.

[31] S. Wang, H. Chen, and W. Shi, "SPAN: A software power analyzer for multicore computer systems," *Sustainable Computing: Informatics and Systems*, vol. 1, no. 1, pp. 23–34, 2011.

[32] C. Isci, A. Buyuktosunoglu, C. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in *IEEE/ACM International Symposium on Microarchitecture*, 2006, pp. 347–358.